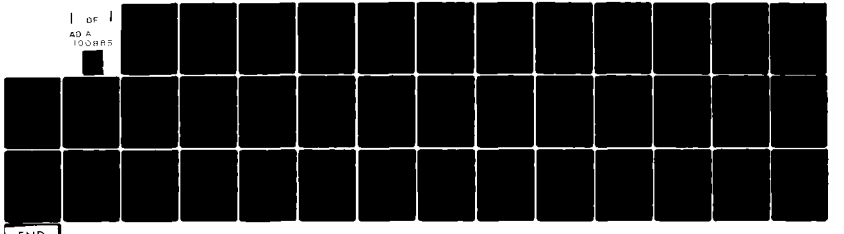
AD-A100 865 KANSAS STATE UNIV MANHATTAN DEPT OF COMPUTER SCIENCE F/G 9/2
PORTABILITY OF OPERATING SYSTEM SOFTWARE.(U)
JUN 81 V E WALLENTINE DAAG29-78-G-0200
UNCLASSIFIED TR-81-05 ARO-16160.7-A-EL NL



END
DATE FILMED
7-81
DTIC

(11)

PORTABILITY OF OPERATING SYSTEM SOFTWARE.

FINAL TECHNICAL REPORT

V. E. WALLENTINE

JUNE 1, 1981

U. S. ARMY RESEARCH OFFICE

DAAG 29-78-G-0200

KANSAS STATE UNIVERSITY

APPROVED FOR PUBLIC RELEASE;
DISTRIBUTION UNLIMITED

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO. AD-A100	3. RECIPIENT'S CATALOG NUMBER 865
4. TITLE (and Subtitle) Portability of Operating System Software		5. TYPE OF REPORT & PERIOD COVERED Final Technical Report Sept. 1979-March 1981
		6. PERFORMING ORG. REPORT NUMBER TR-81-05
7. AUTHOR(s) V. E. Wallentine		8. CONTRACT OR GRANT NUMBER(s) DAAG 29-78-G-0200
9. PERFORMING ORGANIZATION NAME AND ADDRESS Department of Computer Science Kansas State University Manhattan, KS 66506		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS U. S. Army Research Office Post Office Box 12211 Research Triangle Park, NC 27709		12. REPORT DATE June 1, 1981
		13. NUMBER OF PAGES 40
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) Army Institute for Research in Management Information Systems and Computer Science Georgia Institute of Technology Atlanta, Georgia		15. SECURITY CLASS. (of this report) Unclassified
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

NA

18. SUPPLEMENTARY NOTES

The view, opinions, and/or findings contained in this report are those of the author(s) and should not be construed as an official Department of the Army position, policy, or decision, unless so designated by other documentation.

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Portable, Adaptable, Network Operating System, Software Configuration.

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

In this document we summarize the results of a study into the adaptability of system software to a computer network. We first discuss the structure of a Network Adaptable Executive - (NADEX) written in concurrent Pascal. We then discuss the performance of NADEX and the impact of concurrent Pascal on the performance of NADEX.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

FINAL TECHNICAL REPORT:

PORTABILITY OF OPERATING SYSTEMS SOFTWARE*

V. E. Wallentine
Principal Investigator

TR-81-05

Computer Science Department
Kansas State University
May 1, 1981

*This research was supported in part by the Army Institute for Research in Management, Information, and Computer Systems under grant number DAAG 29-78-G-0200 from the Army Research Office.

Table of Contents

		Page
1.0	Research Objectives	1
2.0	Research Results	4
	2.1 Introductory NADEX Concepts	4
	2.2 Adaptability of NADEX Layers	10
	2.3 Performance of NADEX	15
3.0	Future Work	19
4.0	Annotated Summary of Papers and Reports	22
	4.1 Papers Published Outside KSU	22
	4.2 KSU Technical Papers	25
5.0	References	34
6.0	Prototype Module Sizes	35
7.0	Participating Scientific Personnel	36

A

1.0 Research Objectives

The objective of this research was to investigate the portability of operating system (OS) software. True portability is interpreted as the ability to run a program unchanged on multiple heterogeneous host machines. The basic heterogeneity of such host computers means that any OS must be changed to accommodate low-level architectural features. Therefore, we chose to study the structure of an adaptable OS which can execute on various machine, operating system, and computer network architectures. This research effort has produced the following results:

1. an adaptable core operating system written in Concurrent Pascal,
2. a network adaptable executive (operating system) which supports distributed programming,
3. a distributed program construction and control system, and
4. performance comparisons of high-level language based operating systems under structural variations.

During this study, we produced three published papers (and one still in preparation), eleven technical reports, and 30,000 lines of Pascal and Concurrent Pascal [5.1] code which constitute a Network ADaptable EXecutive (NADEX) and its program development subsystems.

In this document we will describe our approach to porting the various layers of an operating system, the basic services and structure of the NADEX systems, and their performance. We will also relate each element of the system to our reports and papers whose abstracts appear in Section 4. In order to distinguish these papers from other references, the other references are in Section 5.

The approach we have taken is to study the functionality and layering of a distributable operating system, define the portability properties of each layer, implement the system (including its program

development subsystems), and then test its performance. The result is NADEX. It has two layers--a distributed programming environment and a core operating system. The core operating system is ported relatively easily due to its implementation in a strongly typed concurrent language--Concurrent Pascal. It provides a message-passing core on which all of the NADEX distributed programming environment is implemented. The portability of the programming environment is based on this ability to pass messages between programs in what we call a software configuration.

These configurations are general graphs of communicating programs (sequential or concurrent) which can be distributed across a computer network. The NADEX distributed programming environment is a software configuration itself; and therefore it is adaptable to other message-based core operating systems (UNIX [5.7], for example) and is distributable across a network. This configuration, in turn, supports the creation, distribution, initiation, synchronization, and termination of distributable user software configurations.

NADEX has been implemented and is running as a prototype on a Perkin-Elmer (Interdata) 8/32. It is ready to be tested in a network environment. In addition, the feasibility of porting it to other core operating systems such as UNIX, UCSD Pascal system, and Perkin-Elmer's OS-32/MT has been studied. All seem to be feasible under varying degrees of effort. UNIX seems to be a willing host while the latter two will involve more effort.

We have also tested the performance of the NADEX core operating system under various structural changes. Performance experiments were carried out which isolated (1) the impact of a centralized buffer system

versus a de-centralized one as the central element of the NADEX core OS, (2) the impact of a high-level language versus an assembler language kernel, and (3) the impact of a multi-level concurrent program (hierarchical virtual Concurrent Pascal machine) kernel.

In Section 2, we overview the layering of the NADEX systems, their relationship to other core operating systems, and the distributed programming tools which have been developed. We also present the results of the performance experiments.

2.0 Results

2.1 Introductory NADEX Concepts

The results of this research are the NADEX core operating system, the NADEX distributed programming environment, and the performance and portability properties of each. This section contains a discussion of the function and layering of the NADEX systems and the impact on the adaptability of this layering. This section concludes with a discussion of NADEX performance measures.

NADEX is a distributed programming environment and a core operating system whose objective is to support modular programming. This concept of "programming in the small" which has been so successful in UNIX [5.7]--in the form of pipelines of communicating sequential processes--is extended to support general graphs of communicating programs under NADEX. These general graphs are called software configurations and consist of nodes which communicate via Data Transfer Streams (DTSs). These DTSs are full-duplex in nature and, therefore, support bi-directional communication between any two nodes which they connect. Nodes access DTSs via ports. These ports are distribution-independent and, therefore, permit nodes of a configuration to be distributed across a computer network without reprogramming.

NADEX supports three programmer views--the single node programming view, the data flow abstraction, and the overall software configuration structure. It provides the compilers and PREFIX for sequential and concurrent programs, the DTS operations, and the configuration descriptor for a distributable configuration, respectively, for these programmer views. It also permits expression of these views in a user tailored system. Command processors, utility subsystems (such as file

systems), and configuration description languages can be specified by the user. These systems can be constructed upon NADEX which provides only the essential elements of an operating system--interprocess communication, a representation for distributable, communicating processes (software configurations), and resource allocation. It is the DTS concept which permits software configurations to be distributed across a computer network controlled by NADEX.

NADEX supports a concept we call a software configuration--referred to as merely a configuration throughout this document. A configuration consists of a collection of nodes connected by data transfer streams (DTS's). Nodes can be user programs (both sequential and concurrent languages such as Sequential Pascal and Concurrent Pascal), file access nodes (for accessing files within the NADEX file system), I/O device access nodes (for accessing I/O devices not supported by the NADEX file system), or external configurations such as subsystems.

Nodes within the configuration are connected by DTS's which are also called connections. Each connection consists of two bi-directional components--data and parameter. The data component transfers data in page-sized blocks (a page is 512 bytes) and interfaces to the user program at the page, logical record, or character level. The parameter component transfers small parameter blocks typically used for control information. The data and parameter components are totally independent. The two directions of each component are independent in the sense that each direction has its own queue, but the user protocol restrictions are defined in terms of the bi-directional components.

For purposes of these discussions, we will speak of a node issuing reads and writes to a DTS. These should be assumed to be read-page and

write-page requests for the data component, and read-parm and write-parm requests for the parameter component. The blocking of character and logical record data into pages is handled by a PREFIX of the nodes and will not be discussed here. Unless otherwise specified, all discussions apply equally to data and parameters, and no distinction will be made.

There are no structural restrictions on the graph formed by the nodes and connections (DTS's). In particular, it need not be linear (as in UNIX [5.7]) or hierarchical. It need not even be acyclic or connected. Nodes are not precluded from having connections to themselves. Thus, the configuration is described by a (labeled) undirected graph. The labeling occurs where each connection enters the two (not necessarily distinct) nodes it connects.

The user programs (as well as the various system routines which implement the other nodes) address the connections emanating from each node by DTS identifiers local to the node. These local DTS identifiers are also called port numbers. The meaning of the data stream associated with each port is defined by the program. Port numbers are generally assigned by the programmer starting with one. These port numbers are the labels on the configuration graph.

The structure of a configuration is defined by an interactive (PCD) construction program which builds a file called a Partial Configuration Descriptor (PCD). The PCD defines the structure of the configuration and the type (user program, file access, etc.) of each node. PCD files can be hierarchical so that they can be constructed in a modular manner as a composition of other PCDs. When the user requests that a configuration be run, via a terminal command language, the PCD is used along with information from the command to construct a configuration

descriptor (CD). The configuration descriptor includes all of the information about the configuration including, for example, the names of the files to be accessed by the file access nodes. The configuration descriptor contains enough information for the system to allocate resources across the network. This set of distributed programming tools which are available under NADEX is illustrated in Figure 1. In addition to the PCD construction tools, the structure of the PCD can be shown graphically as well as textually as seen in Figure 1. Thus, the PCD Workbench provides the user with the ability to specify general graphs of programs in a terminal command language, construct them in a hierarchical manner and automatically generate graphic and text output. The set of programs and tools of Figure 1 are documented in reports 4.2.4 through 4.2.8. If the NADEX programming system is used on a single machine, the configuration is submitted to the NADEX core operating system for execution.

The PCD Workbench programs permit the user to specify the manner in which the configuration is to be distributed across a computer network. Thus, the CD contains information about the residence of data and programs within the network. Given this Network Configuration Descriptor (NCD), the NADEX distributed programming environment can distribute the configuration across the network; and it can initiate, synchronize, and terminate the individual parts of the configuration.

The manner in which this is accomplished is illustrated in Figure 2. The NCD is broken-up into the parts (called local CDs) which are to be run on the separate machines. The LCDs are sent to a network configuration control (NCC) program on each machine. The NCC programs across the network thus cooperate to synchronize the initiation,

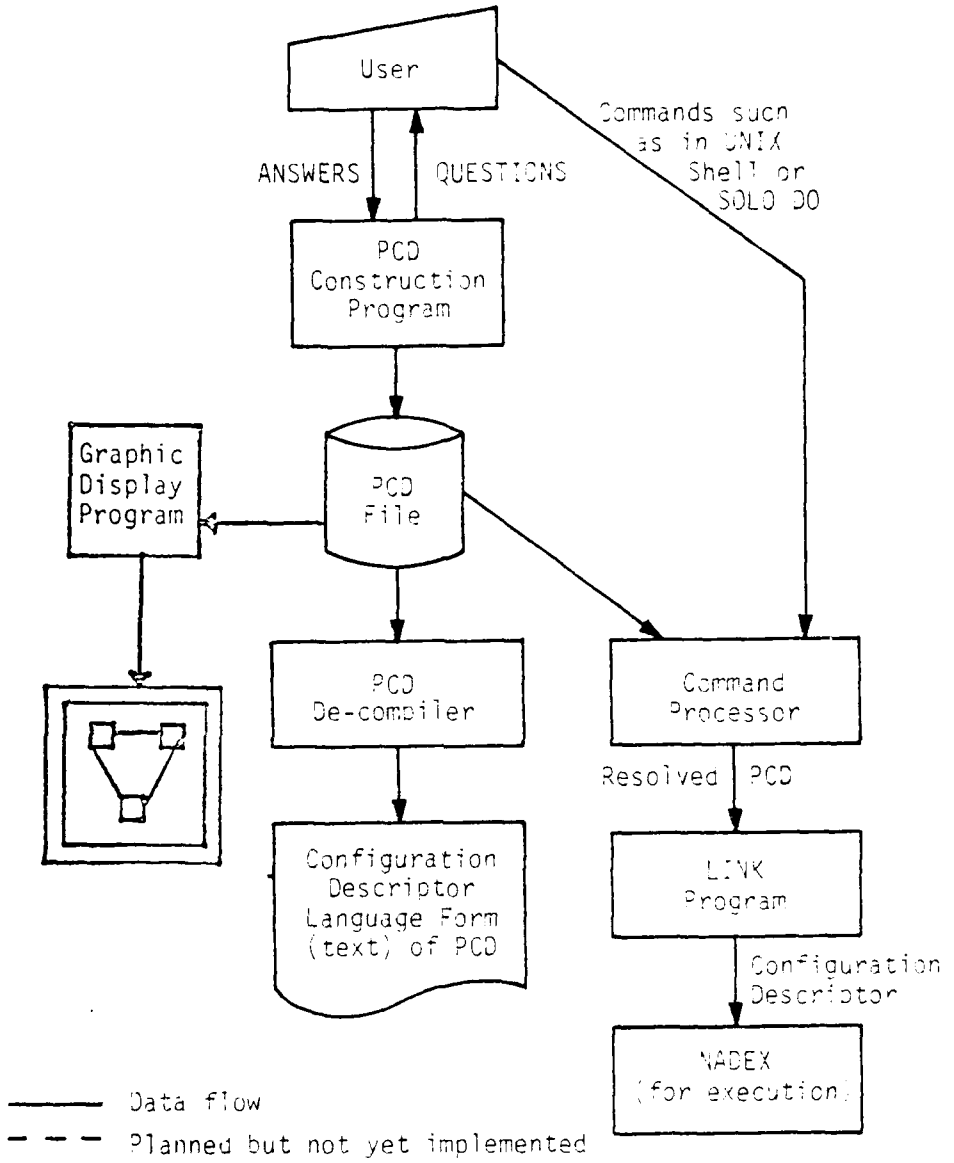


Figure 1. The PCD Workbench

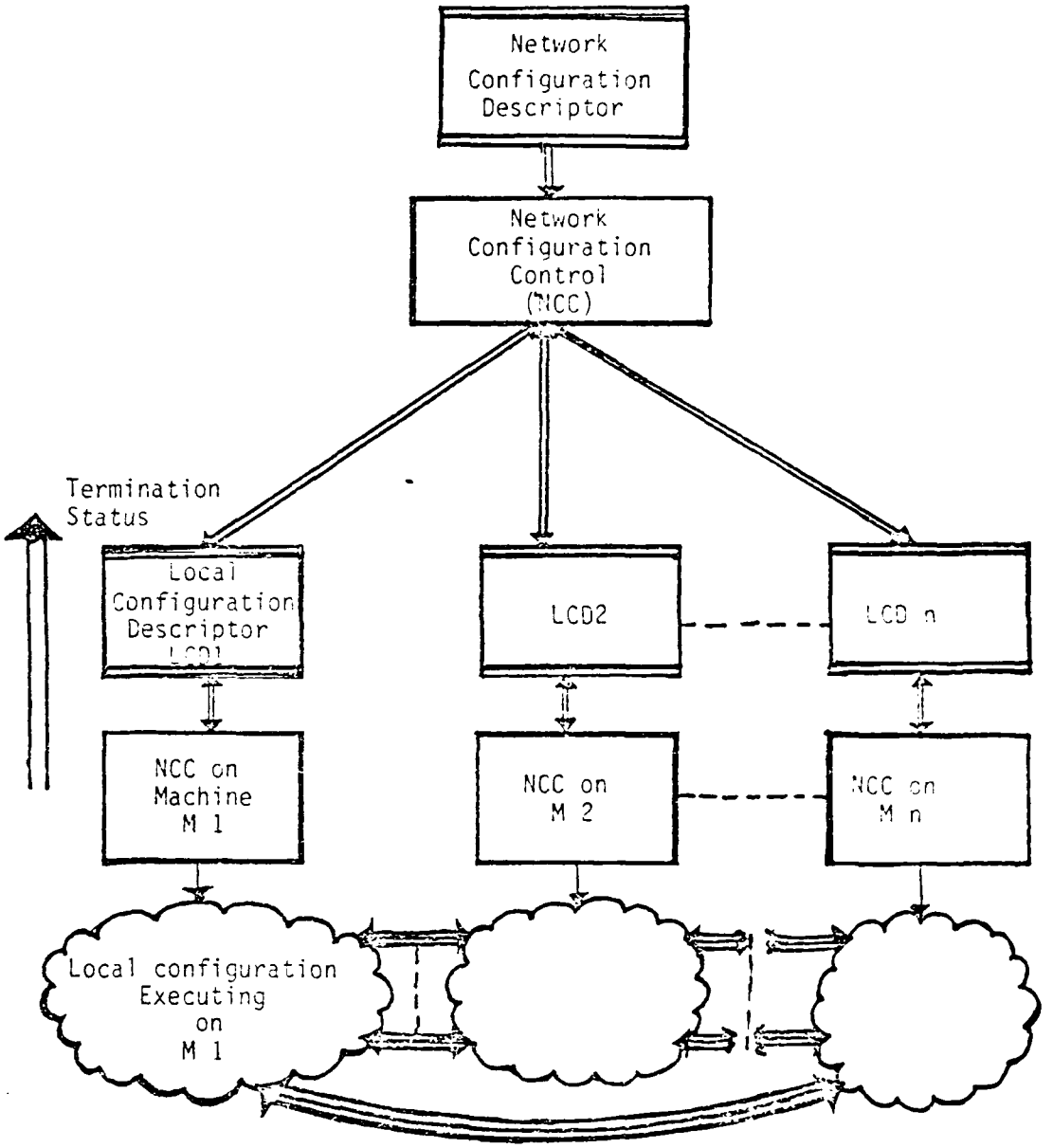


FIGURE 2

DISTRIBUTION OF SOFTWARE CONFIGURATIONS ACROSS A COMPUTER NETWORK

execution, and termination of distributed configurations. The execution of an LCD is carried out by a local NADEX or other core operating. The implementation of the data exchange between nodes (DTSs) is via a network inter-process communication system called MIMICS. More detail on the components of each element of NADEX is presented in the next section.

2.2 Adaptability of NADEX Layers

Figure 3 contains an illustration of the implementation structure of the NADEX distributed programming environment and its relationship to the NADEX core operating system and the UNIX core operating system. In the remainder of this section, the functions of the components of this system will be discussed. The porting properties of NADEX will also be discussed and these will be illustrated in Figure 4.

The NADEX core operating system is responsible for execution of local configuration descriptors. The properties of this core operating system are described in reports 4.2.1 through 4.2.3. Given that UNIX supports message-passing between processes, it could be used as the core operating system. However, as described in report 4.2.10 UNIX needs additional facilities to support multi-user subsystems. The PORTS facility of Sunshine [5.6] and Zucker [5.8] make it suitable as a core operating system onto which the NADEX distributed programming environment (DPE) can be adapted.

The command processors (MIRC, UNIX, and DO), the hierarchical file subsystem (HFSS), the link program (LINK), the network file subsystem (NFSS), the SUBMIT node, the network configuration control (NCC), and

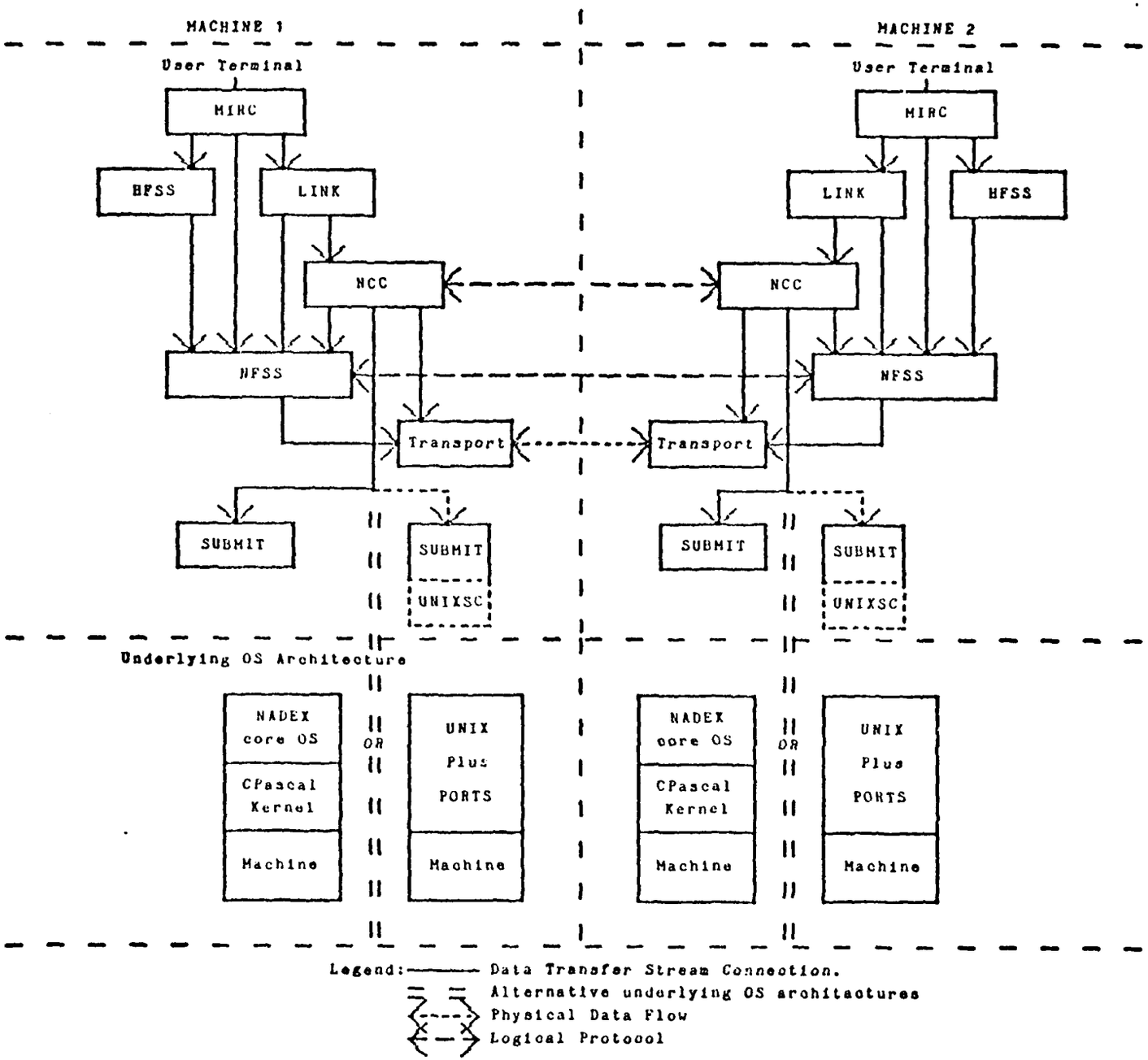


Figure 3. The Distributed form of NADEX

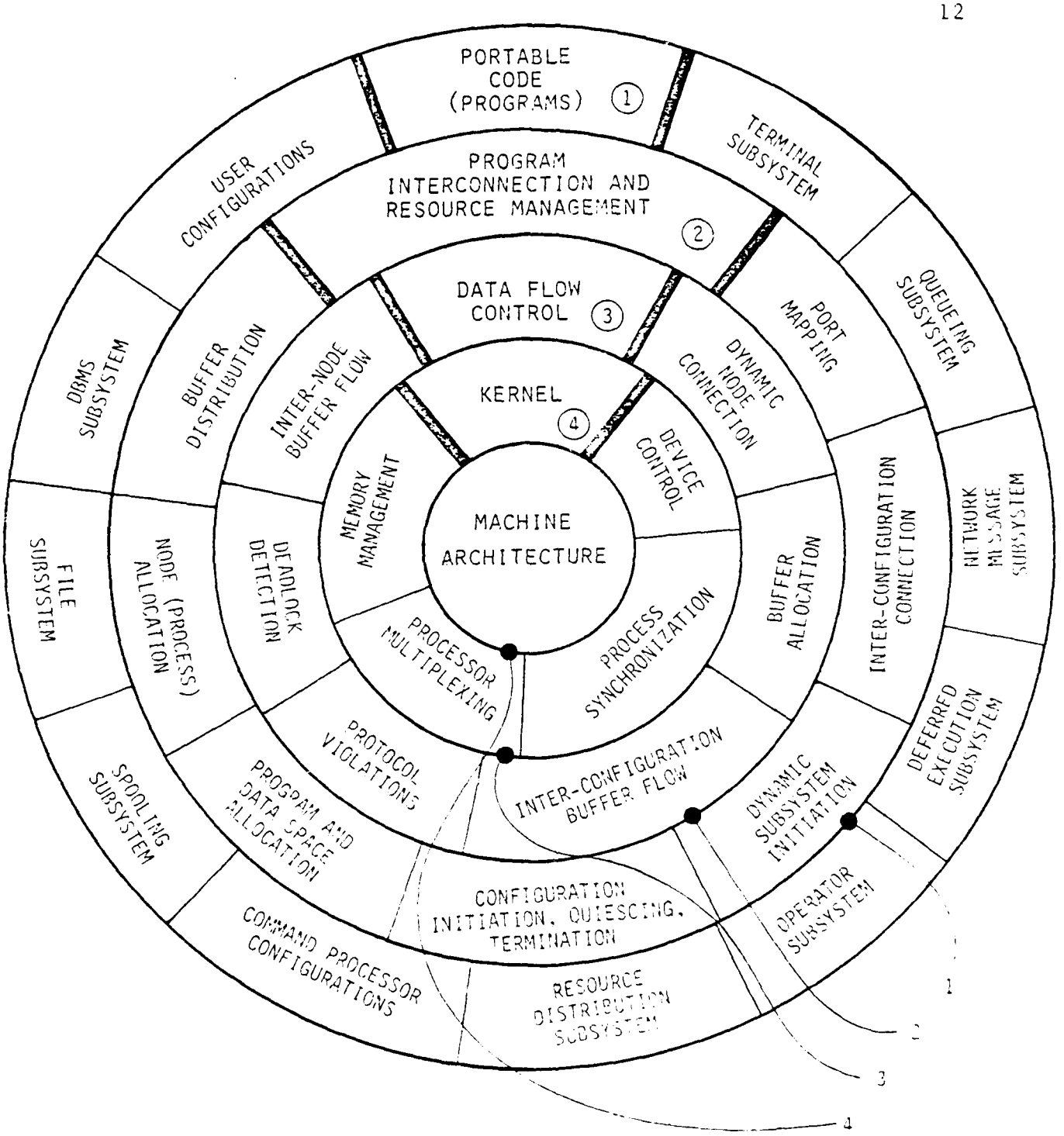


FIGURE 4

NADEX STRUCTURE

the transport level (XMS-message system) are all programs in network-wide software configuration. The command processors, NCC and LINK programs have already been discussed, and the HFSS is a prototype UNIX file system. The SUBMIT node submits a local configuration descriptor to the local core operating system and returns its completion code to NCC. The NFSSs communicate across the network to control files across the network. Finally, the transport service (MIMICS) moves data between ports on separate machines.

In Figure 4 there is a view of the implementation structure of the NADEX core operating system. We will refer to this figure in describing the adaptability properties of NADEX. The outer ring (1) of the onion consists of configurations. The NADEX DPE resides at this level. Therefore, both the NADEX core operating system and UNIX with PORTS will support the NADEX DPE. As shown in Figure 3, a small map between data transfer streams and pipes in UNIX must be added to SUBMIT to adapt the NADEX DPE to UNIX. (This map has been programmed but not debugged due to lack of a good UNIX system available.)

The closer one gets to the center of the onion, the more machine and/or operating system dependent an adaptation of NADEX DPE becomes. Since all of NADEX DPE was written in Sequential Pascal, it adapts to message-based core operating systems very well. Only a Pascal compiler and a SUBMIT map are needed. This level of effort is on the order of one man-month. Furthermore, the NADEX core operating system (rings 2 and 3) are written in Concurrent Pascal and, therefore, port directly to any system which supports hierarchical Concurrent Pascal programs. Ring 4 is the Concurrent Pascal kernel. It is written in Sequential Pascal and ports nicely to P-code type machines such as the Western Digital

Pascal Microengine.

Four feasibility studies were carried out under this research effort to ascertain the effort necessary to adapt the NADEX DPE to several machine and operating system environments. It is currently running on a Perkin-Elmer (Interdata) 8/32. The first experiment was to host it on a UNIX system. It took about one month's effort to code. The second experiment involved studying the Concurrent Pascal compiler to ascertain the effort to generate P-code for the Western Digital. This took about four (4) months effort in its preliminary work. It would take another two months to complete the task--total of six months effort. This study is documented in report 4.2.11.

The objective of the third and fourth feasibility studies was to assess the difficulty of integrating the facilities of the NADEX core OS into existing core operating systems. The operating systems chosen were Perkin-Elmer's OS-32/MT and the UCSD Pascal P-system. The approach was to use the kernel facilities (ring 4) of each OS and then integrate rings 2 and 3 code of NADEX into these operating systems. We programmed ring 3 for both machines--in assembly language for the Perkin-Elmer operating system and in UCSD Pascal for the other. In both cases, it took two month's effort each. Ring 3 seems to be about twice as complex. Therefore, assuming that memory and other resources are enough to hold NADEX, six months' effort is a good estimate of the total porting effort per system. Of course, the rate of generation and understanding of this level of code is extremely sensitive to the quality of the programmer.

2.3 Performance of NADEX

The study of the performance of NADEX was carried out in two phases. The first phase was to instrument NADEX and record the percentage of run-time that NADEX spent in each ring. The second phase involved recoding rings 3 and 4 and comparing the performance of the new and old versions. Table I contains the percentage of time in each ring for a three-node configuration where the nodes only pass messages between each other and do no real computation. The objective is to isolate the elements of the core OS which can be improved in speed and get the most improvement in overall performance of NADEX.

	% time spent
RING 4	29.75
RING 3	15.00
RING 2	40.00
RING 1	15.25

Table I

It is clear from Table I that the critical factors to performance are the kernel and the centralized buffer (data flow) manager. In the original form of NADEX, the data (pipeline) buffer manager (called the PBM) was written as a decentralized manager so that contention for buffers would not be a bottleneck if the core OS were to be run on a multiprocessor with shared memory. In order to test the overhead of this structure, we recoded the PBM into a centralized version and tested the performance of both against the performance of an assembler buffer manager in OS-32/MT--a representative of current day operating systems. Table II gives the time to transfer one message between two nodes using all three modules.

Thus, it is clear that the centralized PBM is superior to both other methods and drastically improves the performance of NADEX; and it is also written in a high-level language.

	Transfer Time
Decentralized	10.0 milliseconds
Centralized	2.3 milliseconds
Assembler	4.0 milliseconds

Table II
PBM Performance

The next most critical element of performance was the kernel of Concurrent Pascal--ring 1. The only degradation in performance that we envisioned was the difference in performance of Pascal vs. assembler language. We chose to code it in assembler as well. The assembler kernel ran only 5 percent faster than the kernel written in Pascal. This was a surprise. We expected an improvement of at least 25 percent. However, it does give an indication that a portable kernel written in Pascal does not incur a substantial performance penalty.

It is interesting to note that the code size of the assembler kernel was 33 percent smaller than the size of the Pascal kernel. Thus, recoding in assembler saves considerable space but does not improve performance substantially. The size of code of each module in NADEX is presented in Section 6.

During the course of this research effort, we developed the concept of a Concurrent Prefix. This is similar to Per Brinch Hansen's Prefix [5.1] for Sequential Pascal except that it permits Concurrent as well as Sequential Pascal programs to be executed and have services to the lower

level Concurrent Pascal program--the OS. In this case, it is NADEX. This provides a true hierarchy of virtual Concurrent Pascal machines. This is illustrated in Figure 5. This virtual machine facility adds significant levels of complexity to the Concurrent Pascal kernel. We measured the performance of both versions of the kernel. We found that the virtual CPascal machine kernel ran 42 percent slower than the non-virtual machine kernel. Thus, the kernel overhead to maintain the hierarchy of virtual machines is a signficant performance factor; and only some microcode or hardware assist will improve the performance.

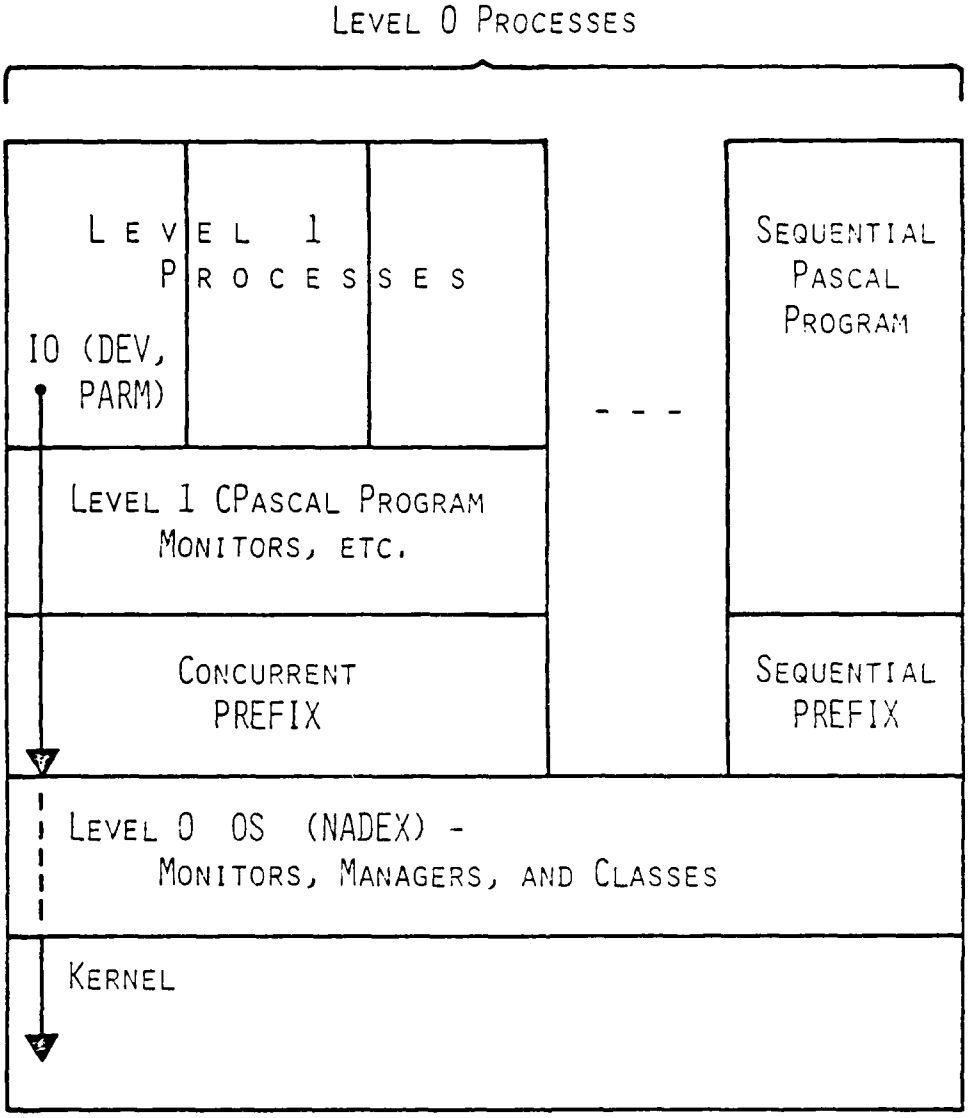


FIGURE 5

VIRTUAL CPASCAL MACHINE

3.0 Future Work

Under this research support, a Network ADaptable EXecutive (NADEX) has been developed which provides a distributed programming environment. This executive is implemented in Concurrent Pascal; the results of performance testing on NADEX established the viability of a highly structured concurrent language for implementing operating systems. The strong typing of Concurrent Pascal provides the relative portability. Extensions to this work fall in two areas: work to improve the utility of the NADEX systems and research into distributed computing for which NADEX is an excellent testbed. In the area of extending the utility of NADEX, a modified version should be developed for personal computers. NADEX should also be recoded into the ADA programming language to improve its portability. In the second area, new distributed programming tools need to be developed. NADEX will serve as an excellent development environment and a good performance test load.

In order for an operating system and its support software to be truly portable, it must be written in a language which has wide acceptance as a standard; and it must have the support of industry, academia, and the federal government to provide compilers for many machines. ADA is such a language. The NADEX distributed programming environment and the NADEX core operating system would be even more portable if written in ADA. This task is relatively straightforward because of the closeness in philosophy and typing of ADA and Concurrent Pascal. The NADEX distributed programming environment could be hosted on any machine for which there is an ADA compiler. The NADEX core operating system could be used to support this environment on bare machines which permit access to low-level devices from ADA. Performance

of these ADA-based systems could then be measured on both high-level language machines (such as the INTEL 432 (ADA) machine and the Western Digital ADA microengine) and on general register machines (such as the DEC VAX-11/780, the Perkin-Elmer 3200 series, and the IBM 370 architectures) for which compilers are now being constructed.

In order for NADEX to be portable to smaller systems (personal computers), a version of NADEX should be implemented for a Limited Capability Host. In this system the small system (LCH) should be able to run local configurations as well submit configurations to a supporting host to distribute across a NADEX-controlled network. This system would provide the personal computer user access to the resources of the network. Small machines such as those running UCSD Pascal or TSI-ADA are great program development tools but sometimes need access to larger machine resources.

Further work needs to be done in the area of run-time specification and validation of inter-program communications protocols. Programs (sequential or concurrent) in software configurations communicate via ports which have low-level properties such as number of buffers. The NADEX core operating system validates only the number of buffers used between programs. However, at the programmer level these ports are data streams. Any protocol between programs exists only in the programmer's mind. A syntax to describe these inter-program protocols is needed so that (1) the programmer can describe them in a descriptive manner, (2) the system can compile these descriptions, and (3) when these programs are combined together into a software configuration, the ordering and typing of messages exchanged between programs can be checked for validity.

The syntax suggested by our research is protocol expressions as documented in our technical report [4.2.4]. Extensions to include predicates in protocol expressions should also be investigated. The basic research to be undertaken would be the syntax, semantics and implementation of a Configuration Description Language which incorporates these inter-program protocol expressions. Properties of this language should include hierarchical encapsulation (packaging) and documentation control. The obvious implementation of this proposed system is via a preprocessor so that it could be modified to adapt to other programming languages such as ADA.

The NADEX distributed programming environment currently needs explicit commands from the user in order to distribute software configurations across a network. This information (resource availability, data program placement, and frequency of program/data use at each site) can automatically be collected during execution of the NADEX environment. Extensions to permit NADEX to collect and use this information would support research, development and experimentation with automatic network-wide resource allocation.

Finally, NADEX performance is strongly affected by the transport level (message system) software complexity. Its performance needs yet to be tested on a high-speed local network. It is hypothesized that the transport layer of software is the greatest bottleneck to good performance. Thus, valuable research in performance of distributed systems would include development of hardware to off-load this transport layer to a microprocessor which interacts with a local network fibre-optic cable. The performance of such a system under NADEX could then be tested.

4.0 Annotated List of Papers and Reports

4.1 Papers Published Outside KSU

4.1.1 V. E. Wallentine, "Experience with Concurrent Pascal as an Implementation Language," Proceedings of the Conference on Microprocessors in DOD, Pingree Park Conference Center, Colorado, August 1979.

Per Brinch Hansen designed and implemented the programming language Concurrent Pascal [5.1] on a PDP 11/45 at the California Institute of Technology. In this paper, we present the basic concepts in Concurrent Pascal and its relationship to Sequential Pascal. Concurrent Pascal has been used at KSU to implement several large systems including a multi-user operating system. We discuss the extensions to the language necessary to implement these systems. Finally, we discuss the problems involved in adapting Concurrent Pascal to several machine environments including microprocessors with small address spaces.

4.1.2 V. E. Wallentine, "Programming Issues in Distributed Systems," Proceedings of the Network IPC Workshop, Georgia Institute of Technology, Atlanta, Georgia, November 1978.

If we are to be successful in distributing programs across highly distributed systems, we must provide the programmer of dynamically interconnected cooperating processes a job control language (software configuration control) as easy to use as Hoare's communicating sequential processes. It seems that the most promising direction is to extend the concept of the UNIX shell to automatically generate the more complex protocols available to the parent processes previously described. It must then also be extended to generate (representations of) distributable configurations of communicating processes. Work in this area is underway at Kansas State University. The project involves

development of a Network ADaptable EXecutive (NADEX). The attempt is to permit the user to specify data flow at the command level and have the command interpreter generate a distributable software configuration of nodes connected by full duplex data transfer stream connections (DTS connections) to form an undirected graph. In general, a node may be thought of as a process. Each of the connections consists of two independent bi-directional data transfer streams. One of these streams uses small parameters while the other uses a standard-sized data buffer. The data buffers carry along with them size and status indicators whereas the parameter buffers contain only a small amount of user-supplied data. In this paper we present a brief overview of the properties of software configurations.

4.1.3 F. J. Maryanski, P. S. Fisher, and V. E. Wallentine, "Data Access in Distributed Data Base Management Systems," Journal of Information and Management, Vol. 2, Number 6, North-Holland Publ. Co., December 1979.

Distributed data base systems have been advocated as the solution to a large number of data processing problems by increasing data accessibility, security, and throughput while reducing cost and resource requirements. Unfortunately, commercially available distributed data base systems have not yet appeared. This paper attempts to provide the potential user or designer of a distributed data base system with an understanding of the basic operational characteristics of such systems. The emphasis is upon the mechanism for data access which is an essential component of any data base system. Our intention is that the reader gain an appreciation of the capabilities and complexities of distributed data base management from the explanation of the data access mechanism.

This paper first discusses the basic structure of distributed data base systems by detailing the functions of the system components. Then in parts three and four, mechanisms are presented for the placement and access of data in a distributed data base system. The fifth part deals with the movement of data among machines and then the sixth section briefly discusses the concept of multiprocessor backend machines. The final portion discusses data integrity considerations in distributed data bases.

4.1.4 V. E. Wallentine and R. A. Young, "NADEX--An Environment for Distributed Programming," In preparation for the Journal of Computer Networks, June 1981.

User access to resources in a computer network has typically taken the form of communicating sequential processes. In such systems, a user process executes an application program; and whenever it needs access to a resource, it sends a message to a system (server) process on some machine in the network which manifests requested resource (for example, a file or device). The server process then accesses the resource and returns a message to the application. Current extant systems [5.2] provide for the distribution of multiple servers for file or device access to communicate to a single application process. Medusa [5.5] and Star OS [5.4] extend this facility to the "task force" concept where the tasks on a tightly coupled set of machines can form a general graph of communicating processes. In this paper, we discuss a distributed programming environment called NADEX (Network ADaptable EXecutive) which supports the distribution of software configurations across loosely coupled networks. These software configurations are general graphs of

programs where each node communicates via ports. This extends the work of Hoare's CSP [5.3] by buffering on the connected ports and by permitting nodes to be concurrent as well as sequential programs. We first present the concepts of software configurations and their use. We conclude with the structure of the NADEX implementation concepts.

4.2 KSU Technical Reports

4.2.1 R. A. Young and V. E. Wallentine, "The NADEX Core Operating System Services, Tech. Rpt. KSU-CS-TR-79-11, February 1979.

NADEX is an operating system whose objective is to support modular programming. This concept of "programming in the small" which has been so successful in UNIX [5.7]--in the form of pipelines of communicating sequential processes--is extended to support general graphs of communicating sequential graphs (CSP) [5.3] under NADEX. These general graphs are called software configurations and consist of nodes which communicate via Data Transfer Streams (DTSs). These DTSs are full-duplex in nature and, therefore, support bi-directional communication between any two nodes which they connect. Nodes access DTSs via ports. These ports are distribution-independent and, therefore, permit nodes of a configuration to be distributed across a computer network without reprogramming.

NADEX supports three programmer views--the single node programming view, the data flow abstraction, and the overall software configuration structure. It provides the compilers and PREFIX for sequential and concurrent programs, the DTS operations, and the configuration descriptor for a distributable configuration, respectively, for these programmer views. It also permits expression of these views in a user

tailored system. Command processors, utility subsystems (such as file systems), and configuration description languages can be specified by the user. These systems can be constructed upon NADEX which provides only the essential elements of an operating system--interprocess communication, a representation for distributable, communicating processes (software configurations), and resource allocation.

It is the DTS concept which permits software configurations to be distributed across a computer network controlled by NADEX. In this document, we briefly describe the structure of NADEX, the basic concepts of software configurations, and the services supplied by the NADEX Core Operating System (Core OS). We assume the reader has knowledge of Sequential and/or Concurrent Pascal [5.1] and a basic knowledge of conventional OS services or a set of desired OS services. Under this assumption, this document contains sufficient material to enable the reader to program normal user programs as well as his or her own command processors and subsystems, such as a file system.

4.2.2 R. A. Young and V. E. Wallentine, "The Structure of the NADEX Operating System," Tech. Rpt. KSU-CS-TR-79-12, November 1979.

NADEX is an acronym for Network ADaptable EXecutive. NADEX supports the building of software configurations which consist of a general graph of communicating nodes. These nodes may be sequential or concurrent programs which access NADEX services through a native PREFIX. The PREFIX concept was originally defined by Per Brinch Hansen as an interface to the SOLO [5.1] operating system. The NADEX Native PREFIX is the interface to the NADEX Core OS and provides data flow abstractions to the program running in a node. These operations permit

each program (running in a node) to exchange messages with other nodes in a software configuration via full-duplex data transfer streams.

In this document, we first present the concept of a software configuration. We then present the general structure of NADEX. Finally, we describe the function of each module of the NADEX Core OS as it is written in Concurrent Pascal [5.1].

4.2.3 R. A. Young and V. E. Wallentine, "Implementation of the Kernel of Concurrent Pascal/32, Tech. Rpt. KSU-CS-TR-79-13, December 1979.

Based on the structured multiprocessing concepts in CPASCAL, we chose CPASCAL as the implementation language for a multi-user operating system called NADEX--Network ADaptable EXecutive. In the design of NADEX, CPASCAL concepts were kept in mind. The dynamic allocation of resources (buffers and memory) required that the manager concept be implemented as an extension to CPASCAL. In order to pass data between processes in an efficient manner, it was decided to add records as system components. Thus, a reduction in the amount of data copying into and out of data encapsuled in classes is achieved. A side benefit of records is that user access to shared data need not be constrained to any particular data encapsulation-entry procedure. The mechanism to achieve the dynamic allocation of these system components to processes is controlled pointers to system components. These pointers, to classes and records, are destructive assignment (even as parameters) so that no new time-dependent error possibilities are introduced into CPASCAL.

A second extension was to introduce hierarchical concurrent programs. This support, in contrast to the first extension which

required only compiler changes, requires extensive kernel support. The kernel must be aware of the multiple levels of concurrent programs. In this document, we discuss the support necessary for multiple levels of processes. A third extension was facilitated by our implementation of the kernel. We coded a portion of the kernel in Sequential Pascal which permits easy modification of the functions within the kernel. Thus, new entry points to the kernel could be required. To solve this problem, we introduced a kernel prefix so that no compiler changes are necessary when new functions are added to or changed in the kernel.

4.2.4 K. L. Rochat and V. E. Wallentine, "A Software System Structuring Tool for Message-based Systems, Tech. Rpt. KSU-CS-TR-80-04, August 1980.

Interest in message-based systems which support software configurations is increasing. A software configuration is a network of processes connected together by ports through which they communicate. The Software Systems Structuring Tool (S^3_T) is an attempt to integrate the common aspects of message-based systems into a software engineering tool for the construction of software configurations.

This tool supports the typed interconnection of modules which allows the verification of the correctness and completeness of interconnection, incremental construction of configurations, and an implementation-independent structure representation. S^3_T uses independently compiled modules with typed ports to construct distribution-independent configurations.

The ability to provide enhanced help information, allow the specification of parameters by position or keyword, and permit the

construction of software configurations using named ports are features which the user will appreciate. In addition, this tool describes the resources needed to execute each module.

This structuring system has been implemented at Kansas State University in conjunction with the NADEX operating system.

4.2.5 R. Fundis and V. E. Wallentine, "Command Processors for Dynamic Control of Software Configurations," Tech. Rpt. KSU-CS-TR-80-02, August 1980.

Command language facilities for the construction and execution of software configurations (networks of communicating processes) are very limited today because current operating systems do not support this level of complexity. The Network ADaptable EXecutive (NADEX) is an operating system which was designed to support dynamic configurations (those configurations which are constructed at command interpretation time) of cooperating processes. These dynamic configurations include arbitrary graphs which may contain cycles. Three command processors have been developed to demonstrate the sufficiency of the NADEX facilities to support dynamic configurations.

NADEX facilities, an overview of the Job Control System, and the command processor configuration environment are presented, followed by user's guides for the command processors. Each command processor has different responsibilities and capabilities for handling configurations. The NADEX static command processor executes completely connected configurations. The UNIX command processor allows linear configurations to be constructed dynamically, and the MIRACLE command processor allows the dynamic construction of arbitrary configurations. Syntax graphs and

sample user sessions are presented for each command processor.

4.2.6 R. L. Rochat and V. E. Wallentine, "NADEX Job Control System Implementation," Tech. Rpt. KSU CS-TR-80-05, July 1980.

NADEX is an operating system whose objective is to support modular programming. This concept of "programming in the small" which has been so successful in UNIX [5.7] (in the form of pipelines of communicating sequential processes) is extended to support general graphs of communicating sequential programs under NADEX. These general graphs are called software configurations and consist of nodes which communicate via Data Transfer Streams (DTSs). These DTSs are full-duplex in nature and, therefore, support bi-directional communication between any two nodes which they connect. Nodes access DTSs via ports. These ports are distribution-independent and, therefore, permit nodes of a configuration to be distributed across a computer network without reprogramming. In this paper the software tools which support the construction of software configurations are described. These tools consist of an interactive partial configuration descriptor (PCD) builder, a PCD decompiler (text formatter), and a linker of nodes (LINK program). They form the basis for the job control system of NADEX.

4.2.7 R. L. Rochat and V. E. Wallentine, "NADEX Utility Programs," Tech. Rpt. KSU-CS-TR-80-06, August 1980.

In this paper two utility programs are described which add capabilities to NADEX. These programs are a hierarchical file system (HFS program) and a program which allows a single console node to be

shared by several nodes of a user's configuration--a console multiplexor (CM program). These programs were developed for use with the dynamic command processors [4.2.5]. The reader is assumed to be acquainted with Sequential and Concurrent Pascal [5.1], the command processor functions [4.2.5], and the services of NADEX [4.2.1].

4.2.8 R. Sanders, "A Graphics Support System for Programming Communicating Processes," Tech. Rpt. KSU-CS-TR-80-01, August 1980.

The complexity of many sophisticated programming tasks requires a methodology to simplify and filter information to a manageable level. The GSS (Graphics Support System) described in this document will draw pictures of software configurations. A configuration is a directed graph containing one to eight nodes. Each node can consist of a sequential or concurrent program, be hierarchical in nature, and can itself be a configuration.

The most important contribution of GSS is the method used to determine the complex relationships that exist between the picture components.

Arbitrary configurations can be decomposed into three distinct types of objects: hangers, pipes, and cycles. The decomposition is accomplished by following and analyzing all of the node connections and constructing patterns of linkage. The purpose of building objects is to define a predictable, repeatable heuristic that will draw pictures in the desired manner. The number of nodes in an object determines the shape of the object. An object's shape is used to select a predefined pattern which defines how the nodes will be drawn relative to the

another. Flow into and out of nodes is studied to determine where they should be placed relative to the picture and relative to other nodes within their parent object.

GSS allows the user to interact in the drawing portions of a picture or will draw the picture without user assistance. GSS does not build configurations. It is meant as a documentation tool that assists in the understanding of a software configuration.

4.2.9 D. E. Eaton and V. E. Wallentine, "Hosting the NADEX Environment on the UNIX Operating System," Tech. Rpt. KSU-CS-TR-81-02, May 1981.

Command language facilities for the construction and execution of software configurations (networks of communicating processes) are very limited today because current operating systems do not support this level of complexity. The Network ADaptable EXecutive (NADEX) was designed to support dynamic configurations (those configurations which are constructed at command interpretation time) of cooperating processes. These dynamic configurations include arbitrary graphs which may contain cycles. The NADEX environment runs on top of the NADEX core operating system. The object of this work is to make the NADEX environment so it will run with the UNIX (a trademark of Bell Laboratories) operating system as its core operating system.

4.2.10 V. E. Wallentine, R. A. Young, K. L. Rochat and D. N. Neal, "NADEX Implementation Notes," Tech. Rpt. KSU-CS-TR-80-07, November 1980.

In this document we describe the implementation details of NADEX. We discuss the basic concepts of software configurations, configuration properties, subsystems, dynamic connection to subsystems, distribution of configurations, data transfer stream (DTS) mapping, and DTS manager. The level of detail provided serves as an introduction to the functioning of the NADEX Core Operating System.

4.2.11 M. E. Litteken, "On the Adaptability of Multipass Compilers to Variants of (Pascal) P-Code Machine Architectures, Tech. Rpt. KSU-CS-TR-81-04, May 1981.

This report is a description of an effort to modify the code-generation portion of a multipass Concurrent Pascal (PASCAL) compiler so that it will produce object code for a different machine. The original compiler generated P-Code (Pascal stack machine code) for a virtual machine. The modified version will generate code for the Pascal microengine--a microcomputer whose instruction set is similar to virtual P-Code. The similarity of instruction sets made it appear that the required modifications would be straightforward, and that the high-level language constructs of Concurrent Pascal (monitors, for example) would easily map onto the microengine's architecture. However, as the project progressed, the architectural differences became a significant obstacle. This report contains a description of the organization and [illegible] of the two machines and detail on [illegible] Concurrent Pascal compiler. It also contains a description of the compilation problems caused by the differences between the machines and how those difficulties were resolved.

5.0 References

5.1 Brinch Hansen, P. The Architecture of Concurrent Programs, Prentice-Hall, 1979.

5.2 Forsdick, A. C., et al. Operating Systems for Computer Networks, Computer, 11, 1 (Jan. 1978), pp 48-57.

5.3 Hoare, C. A. R., Communicating Sequential Processes, Comm. of ACM, 21, 8, (August 1978), pp 666-677.

5.4 Jones, A. K., et al. Star OS, a Multiprocessor Operating System for the Support of Task Forces. Proc. 7th Symp. Operating Systems Principles, SIGOPS, 1979, pp 315-330.

5.5 Ousterhout, J. K., et al. Medusa: An Experiment in Distributed Operating System Structure, Comm. of ACM, 23, 2 (Feb. 1980), pp 92-105.

5.6 Sunshine, C., Interprocess Communication Extensions for the UNIX Operating System: I. Design Considerations, Rand Tech. Report R-2064/1-AF, June 1977.

5.7 Thompson, K. and Ritchie, D. M., The UNIX Time-Sharing System, Comm. ACM, 17, 7, (July 1974), pp 365-375.

5.8 Zucker, S., Interprocess Communication Extensions for the UNIX Operating System: II. Implementation, Rand Tech. Report R-2064/2-AF, June 1977.

6.0 Prototype Module Sizes

Module	Lines of Pascal Code	Lines of Assembler Code
NADEX Core OS	4633	541
Centralized PBM	3053	
Command Processors		
NIRC	3052	
UNIX	2268	
DC	2277	
Linker		
LIN	1368	
File Systems		
Hierarchical File System	1113	
Network File System	1662	
Local File System	826	
Network Control		
Network Configuration	1769	
Transport (MINICS)	1462	
Utilities		
Graphics Support	4212	
PCD Builder	2311	
PCD Decompiler	947	
Console Subsystem	830	
Kernel	1818	1330
Library		4000
Total	33,873	5871

7.0 Participating Scientific Personnel

Dr. V. E. Wallentine		
Richard McBride	-	Ph.D., Aug. 1980
Kim Rochat	-	M.S., Aug. 1980
Roxanne Fundis	-	M.S., Aug. 1980
Richard Sanders	-	M.S., Aug. 1980
Denis Eaton	-	M.S., Jan. 1981
Mark Litteken	-	M.S., Jan. 1981
Robert Young		
Patrick Ireland		
Greg Dietrich		
Kim Janne		

END

DATE
FILMED

7-81

DTIC